1979-05

An Experiment in Software Error Data Collection and Analysis

Schneidewind, N.F.

IEEE

IEEE Transactions on Software Engineering, Vol. SE-5, No. 3, May 1979.

An Experiment in Software Error Data Collection and Analysis

N. F. SCHNEIDEWIND, SENIOR MEMBER, IEEE, AND HEINZ-MICHAEL HOFFMANN

Abstract—**The propensity to make programming errors and the rates of error detection and correction are dependent on program complexity. Knowledge of these relationships can be used to avoid error-prone structures in software design and to devise a testing strategy which is based on anticipated difficulty of error detection and correction. An experiment in software error data collection and analysis was conducted in order to study these relationships under conditions where the error data could be carefully defined and collected. Several complexity measures which can be defined in terms of the directed graph representation of a program, such as cyclomatic number, were analyzed with respect to the following error characteristics: errors found, time between error detections, and error correction time. Significant relationships were found between complexity measures and error characteristics. The meaning of directed graph structural properties in terms of the complexity of the programming and testing tasks was examined.**

Index Terms—**Abstract data types, clusters for the implementation of data abstraction, dependency among clusters, dialogues, efficient code for very high level languages, programming languages, programming methodology, set languages, very high level languages.**

INTRODUCTION

THE COMPUTER industry is searching for ways to produce reliable software and to reduce the cost of software development and maintenance [1]. Structured programming has received wide acclaim as one answer to this search [2]. If it can be shown that structural characteristics of computer programs are related to the difficulty of producing programs with few errors and to the difficulty of detecting errors during debugging and testing, then structural characteristics could be used as measures of program complexity for program design and testing purposes. Program designs with poor structural properties would be avoided because their use would likely result in many programming errors and great difficulty of error detection during debugging and testing. Secondly, after programs have been designed and coded, the complexity measures would be used to index the difficulty of debugging and testing the programs. Resources would be allocated to testing in accordance with the expected difficulty of error detection.

The purpose of the research reported in this paper was to test our hypothesis that program structure has a significant effect on error making, detection, and correction as measured by various software error characteristics, such as the number of errors found and labor time required to detect and correct errors. We will say that a significant effect exists if quantitative analysis demonstrates that complex structures should be avoided because of high error occurrence. For this purpose, structure is synonymous with various complexity measures which can be obtained from a directed graph representation of a computer program. If relationships do exist, we wish to determine which complexity measures are best for indicating the likelihood of error commission during programming and the difficulty of error detection and correction during debugging or testing. This work is a continuation of earlier work which has been reported in the literature [3]-[5].

While the Naval Tactical Data System software error data, which were previously used in this project, had the desirable property of being collected from a large-scale tactical system involving the integration of many modules, the following information was not available: 1) error detection and correction labor times, 2) error locations in program structures, and 3) causes of software errors. In addition, large module size and an unstructured programming style (programming took place prior to the advent of structured programming) made error and structure analysis very difficult. Consequently, an experiment in software error data collection and analysis [6] was conducted at the Naval Postgraduate School in order to collect error data with the desired characteristics and to obtain a better understanding of the factors which induce programming errors and affect error detection and correction.

SELECTED EXAMPLES OF PREVIOUS WORK IN SOFTWARE ERROR ANALYSIS

Before describing our experiment in software error analysis, we briefly review selected examples of previous work in this field. Unfortunately, the diversity of programming environments in which software error data have been collected does not permit meaningful comparisons between projects. In each case the results must be interpreted with respect to the mission and operating conditions of the project. A second difficulty is that, although software reliability models abound, there has been a scarcity of projects which have recorded software error data in sufficient detail to be usable for analysis purposes. In particular, there have been few empirical studies which stress the relationship between program structure and the occurrence of errors—the primary subject of this paper. Where comparisons can be made, they appear in a later section.

Endres analyzed errors in systems programs—specifically, errors resulting from the maintenance of DOS/VS (Release 28) developed in IBM Laboratories, Boeblingen, Germany [7].

Manuscript received September 15, 1978; revised January 8, 1979. This research was supported in part by the Naval Air Development Center, Warminster, PA.

N. F. Schneidewind is with the Naval Postgraduate School, Monterey, CA 93940.

H.-M. Hoffmann is with the Federal German Navy, Philipp-Lassen-Koppel 38, 2390 Flensburg, Germany.

The programming involved making the following modifications to the operating system:

1) support of virtual storage,
2) increase of partitions from 3 to 5, including variable partition priority,
3) support of new card I/O devices,
4) support of CRT operator console,
5) smaller extensions (catalogued procedures, timer per partition, adaptation for virtual storage access method),
6) adaptation of the spooling system to the above changes.

This activity involved changes to 500 modules (480 lines per module, typically); 250K instructions and comments were involved. A typical activity involved changing 50 instructions in a 200 instruction module. During a five-month test period 432 program errors were discovered.

A summary of the test results follows.

1) Eighty-five percent of the errors were corrected by changing one module.

2) Forty-eight percent of the changed or new modules had one or more errors.

3) There was an average of 4.8 errors per 1K instructions for new code.

4) There was an average of 7.9 errors per 1K instructions for changed code.

5) Forty-six percent of the errors were attributed to lack of understanding of the machine architecture.

6) Thirty-eight percent of the errors were attributed to implementation problems (initialization, addressing).

7) Sixteen percent of the errors were clerical (spelling).

The qualitative assessment of the error results were as follows.

1) About one-half of the errors would have been curable with better programming techniques: languages, test tools, etc.

2) The other one-half were not related to programming and would have been curable by better methods of problem definition and better understanding of systems concepts.

3) The successful implementation of operating system software is heavily dependent on the computer architecture and configuration details.

4) The dynamic nature of operating system development contributes significantly to error making.

Rubey and others analyzed the errors from a number of small (32K instructions, typically), real-time programs for eleven validation efforts [8]. A particular objective of this study was to compare error occurrences in the program development and validation phases; additionally, the cost of error finding and correction was compared for the two phases. The major results of the study are as follows.

1) There was one source statement in error per ten machine instructions during program development.

2) There was one source statement in error per five-hundred machine instructions during validation.

3) Ninety-eight percent of the errors were found during program development.

4) Two percent of the errors were found during validation.

5) Most of the errors (fifty percent) were attributed to specification problems (incomplete, erroneous, and deviation).

6) Of the serious errors (program terminated prematurely or produced an incorrect result), the most frequent involved decision and sequencing logic instructions.

7) With respect to validation methods, it was found that nonexecution validation methods (tools such as cross-reference listings, automatic flow charters, etc., which do not involve executing the program) were able to find errors earlier (one-half of the errors found by nonexecutable methods were found during the first thirty percent of the validation effort). On the other hand, executable methods found more errors, but over half of these were not detected until the last forty percent of the validation effort.

8) The most significant validation cost factor was program size.

9) Satisfaction of specification does not guarantee acceptability of program because about fifty percent of the errors were attributed to specification problems.

10) It is essential to find errors early because late detection is relatively costly.

Wolverton, in his detailed presentation of a costing methodology, indicated that the "40-20-40" rule has held for a number of large command and control projects [9]. This refers to the labor allocation for design and analysis, code and debug, and check-out and test, respectively. This pattern has been the case in the SAGE, NTDS, GEMINI, and SATURN V projects.

In one of the few articles which discusses the quantification of program complexity, McCabe proposes the cyclomatic number of a program's graph as a measure of complexity [10]. He vividly shows the qualitative relationship between complexity and cyclomatic number by presenting the directed graphs of actual programs in increasing cyclomatic number order. He proposes a threshold of cyclomatic number to be used for controlling program complexity in a software development organization. McCabe's work is a cornerstone of the research reported herein because it identified a complexity measure which was found to have a significant relationship with error occurrence.

Software Error Analysis Experiment

Four projects were programmed by the same programmer in Algol W for execution on the IBM360/67. Project characteristics are shown in Table I. Prior to starting the projects, types of software errors were defined and categorized. Error categories are shown in Table II; error definitions are too lengthy to present here. In addition, worksheets (Appendix A) were designed for collecting detailed information about the design, coding, debugging, and testing phases of each project. Error listings (Appendix B) were made which record the characteristics of each error discovered during the projects. Explanatory comments (Appendix C) were recorded about the nature of the errors. For all project activities, records were kept of clock time and labor time so that information could be obtained about the distribution of labor time to the project phases and the labor time required to detect and correct errors. A set of completed work sheets for even one project is too voluminous to include here. Instead, excerpts from Project 2 are shown in Appendixes A, B, and C.

TABLE I
PROJECT CHARACTERISTICS

Project Type	No. of Source Statements	Man-Hours					Operating System
		Program Design	Coding	Debugging	Testing	Total	
1. String Processing	141	5.0	7.0	4.0	5.8	21.8	OS/MVT
2. Directed Graph Analysis	712	31.0	26.0	55.0	13.0	125.0	OS/MVT and CP/CMS
3. Addition* to Project 2	70	7.0	4.0	3.0	19.0	33.0	OS/MVT and CP/CMS
4. Data Base	1084	24.0	24.5	41.5	11.0	101.0	CP/CMS

*Addition of reachability and reachability index computations to directed graph program.

TABLE II
ERROR CATEGORIES AND TYPES

1. Design Errors
   The following types of errors apply to both categories "System Design Errors" and "Program Design Errors":
   - D1 : Communication Error
   - D2 : Design Negligence
   - D3 : Forgotten Cases or Steps
   - D4 : Timing Problems
   - D5 : Errors in I/O Concepts
   - D6 : Data Design Error
   - D7 : Initialization Error
   - D8 : Inadequate Checking
   - D9 : Extreme Conditions Neglected
   - D10: Sequencing Error
   - D11: Indexing Error
   - D12: Loop Control Errors
   - D13: Misuse of Boolean Expression
   - D14: Mathematical Error
   - D15: Representation Error
   - D16: Misunderstanding of Problem Specifications
   - D17: Other Design Errors

2. Coding Errors
   - C1 : Misunderstanding of Design
   - C2 : Negligence
   - C3 : I/O Format Error
   - C4 : Misplaced Data Declaration
   - C5 : Multiple Data Declarations
   - C6 : Missing Data Declaration
   - C7 : Inadequate Data
   - C8 : Initialization Error
   - C9 : Error in Parameter Passing
   - C10: Inadequate or Forgotten Checking
   - C11: Level Problems
   - C12: Missing Declarations of Block Limits
   - C13: Case Selection Error
   - C14: GO TO Problems
   - C15: Comment Error
   - C16: Forgotten Delimiter
   - C17: Inconsistency in Naming
   - C18: Wrong Use of Nested IF Statements
   - C19: Indexing Error
   - C20: Inconsistent Use of Variables or Data
   - C21: Sequencing Error
   - C22: Flag Usage Problems
   - C23: Syntax Error
   - C24: Loop Control Error
   - C25: Incorrect Exit for Subroutines
   - C26: Language Usage Problems
   - C27: Forgotten Statements
   - C28: Representation Error
   - C29: Control Sequence Error
   - C30: Incorrect Subroutine Usage
   - C31: Other Coding Errors

3. Clerical Errors
   - A1 : Manual Error
   - A2 : Mental Error
   - A3 : Procedural Error
   - A4 : Other Clerical Errors

4. Debugging Errors
   - B1 : Inappropriate Use of Debugging Tools
   - B2 : Insufficient or Inappropriate Selection of Test Cases or Test Data
   - B3 : Misinterpretation of Debugging Results
   - B4 : Misinterpretation of Error Source
   - B5 : Negligence
   - B6 : Other Debugging Errors

5. Testing Errors
   - T1 : Inadequate Test Case(s) or Test Data
   - T2 : Misinterpretation of Test Results
   - T3 : Misinterpretation of Program Specification
   - T4 : Negligence
   - T5 : Other Testing Errors

TABLE III
DEFINITION OF TERMS

Complexity Measures	
N_p	Number of Paths (minimum number of paths: no loop traversed more than once in succession)
V	Cyclomatic Number*: number of independent circuits = number of arcs - number of nodes + 2
R	Reachability: summation, over the nodes, of number of ways of reaching a node
r	Average Reachability: R/nodes
S	Number of Source Statements
Error Properties	
e	Number of Errors Found in Program
T_f	Labor Time Required to Find Errors (Since Previous Error Detection)
T_c	Labor Time Required to Correct Error

Note: All of the above are with respect to a single program.
*The definition of V includes an implicit arc connecting the start and terminal nodes (strongly connected graph).

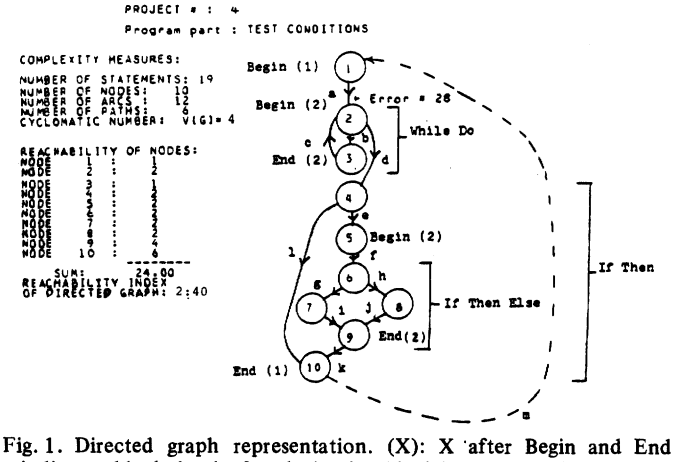

Fig. 1. Directed graph representation. (X): X after Begin and End indicates block level of code in the Algol language. The numbers increase as the indentation of a block moves to the right.

Although the projects did not involve significant requirements analysis or system integration, major contributors to software errors, this lack of realism is more than compensated for by the high quality of the error data. This quality was achieved by taking pains to define error types and to record information about the errors in great detail.

In addition to recording error information, complexity measures, as defined in Table III, were determined for each structure (procedure) by the use of the Project 2 directed graph program and were then related to the error data.

COMPLEXITY MEASURES

Complexity measures were derived from the structural characteristics of the directed graph representation of a computer program. An example of a structure from Project 4 is shown in Fig. 1. Included in the figure are computer printouts of complexity measures obtained from the directed graph program. Decision statements and merge points are represented by nodes (vertices). Arcs represent statements between decision or merge points. They may also be used to show transfer of control. Several measures of complexity, as defined in Table III, can be derived from a directed graph representation of a program, such as the number of paths, cyclomatic number, and reachability. Program size, as determined by the number of source statements, was also used as a complexity measure.

The purpose of the complexity measures analysis was to find measures which relate significantly with the number of errors made or detected in programs and the labor time required for error detection and correction. As used here, a path is a unique sequence of arcs (as identified by the arc's tail and head nodes) from the start node to a terminal node. Paths represent the number of possible ways in which the program can be executed. Thus it would appear that programs with a large number of paths N_p would be difficult to design and debug. In general, this is true. However, as shown by McCabe [10], the number of paths is an inadequate measure of complexity when loops[1] (signifying iteration) exist in a program, such as loop 2, 3, 2 in Fig. 1.

In this case the definition of a path is ambiguous, and in certain cases N_p will be very large or infinite. The possibility exists for a program to traverse a loop any number of times. Unless the definition of path is restricted, each new loop traversal will generate a new path. The additional loop traversals do not produce additional information about program complexity. For this reason, our definition of N_p excludes paths which contain loops traversed two or more times in succession.

McCabe uses the cyclomatic number V to provide an equivalence between independent circuits (one which is not a linear combination of two or more other circuits) and independent paths. One way this is accomplished is by adding an arc from the terminal to the start node (such as the arc from 10 to 1 in Fig. 1). Now one can view the path 1, 2, 4, 5, 6, 7, 9, 10 as being equivalent to the circuit 1, 2, 4, 5, 6, 7, 9, 10, 1. This interpretation of V as the number of independent paths is only valid for single entry and exit programs. The second facet of making independent circuits equivalent to independent paths is to linearly combine independent circuits to generate independent paths. Then the set of independent paths can be linearly combined to generate any path.

Rather than equate independent circuits with paths, we have found a different interpretation of the cyclomatic number to be more useful. Since V is equal to the number of independent circuits, it is equal to a set of substructures which can be identified in a directed graph. When structured programming techniques are used, the independent circuits are identified with the constructs of structured programming: While Do, If Then, If Then Else, etc. This concept is shown in Figs. 1

[1] In the terminology of graphs, a loop is a single arc having the same tail and head nodes. Here "loop" is used in the sense of program iteration.

and 2. In addition, the use of an arc connecting nodes 10 and 1 provides a circuit which can be considered the main line substructure (Fig. 2). Thus, with V = 4 in Fig. 1, the four independent circuits (substructures) are While Do, If Then Else, If Then, and Main Line. The tree or spanning tree shown in Fig. 2 is a connected subgraph which connects all nodes but has no circuits. It has (nodes - 1) number of arcs, which are called branches. Independent circuits are formed by adding an arc at a time from the remaining arcs (called chords). When the circuits are related to a specific tree, they are called fundamental circuits. For the tree, V = 0; it has minimum complexity. Complexity is increased as each chord is added to the tree to form a circuit. These are usually, but not always, transfer of control arcs (c, j, l, m). The tree may be considered the backbone of the structure from which the substructures are formed; it is similar to the main line substructure. This interpretation of V is useful because it gives the number of substructures which must be coded and tested. The greater the number of substructures the higher the program complexity. As shown later, structural characteristics can be used to indicate the relative importance of testing various arcs.

A useful matrix representation of the circuits is the fundamental circuit matrix [11] formed as follows.

1) Rows represent circuits, and columns indicate whether a given arc is part of a circuit.

2) After establishing a flow in the direction of the chord in the circuit, a "1" in a circuit indicates the arc has the same direction as the flow; a "-1" has the opposite meaning; a zero indicates the arc is not present in the circuit.

3) Chords are listed on the left and form a unit matrix (because there is only one chord in a fundamental circuit); branches are listed on the right.

The fundamental circuit matrix for Fig. 1, obtained by using a flow in the direction of the chord, appears below.

	Chords				Branches									
ckt.	c	j	l	m	a	b	d	e	f	g	h	i	k	
C_1	1	0	0	0	0	1	0	0	0	0	0	0	0	While Do
C_2	0	1	0	0	0	0	0	0	0	-1	1	-1	0	If Then Else
C_3	0	0	1	0	0	0	0	-1	-1	-1	0	-1	-1	If Then
C_4	0	0	0	1	1	0	1	1	1	1	0	1	1	Main Line

When a graph is first analyzed, the identity of independent circuits may not be obvious. We may proceed by listing circuits in the above format (without the chord and branch segregation) in order to ascertain whether a circuit is linearly independent. We would identify V of these fundamental circuits in the set which will form a V X V unit matrix. The arcs of the unit matrix are the chord set; the remaining arcs constitute the branches of a tree. Modulo two addition, ignoring signs, of fundamental circuits will produce either an additional circuit or an edge disjoint union (no edge in common) of circuits [11]. The latter is of no interest for our purposes. However, the former could be useful for generating additional circuits which should be tested. For example, the addition of C_3 and C_4 will generate the circuit a, d, l, m. This is important for testing because by generating all circuits from the fundamental circuits, the different execution sequences which must be tested can be identified. Secondly, the frequency of occurrence of an arc in the circuits indicates the relative importance of testing the arc.

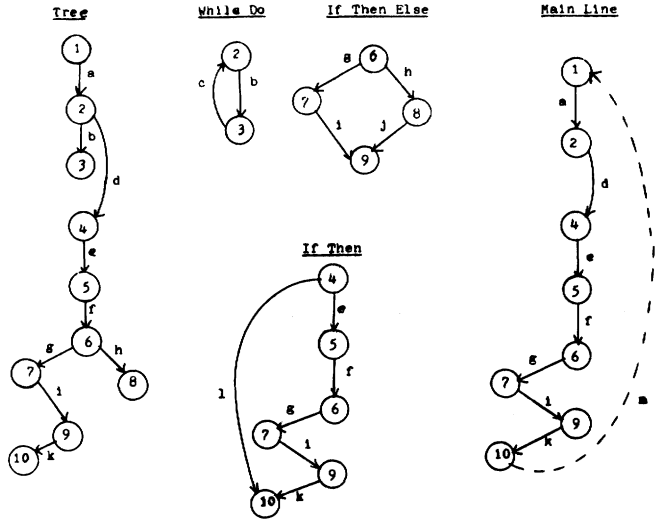

Fig. 2. Decomposition of Fig. 1 structure.

Other complexity measures shown in Fig. 1 which are of interest are the number of paths and reachability. Remembering the restriction on counting paths, that a loop will not be traversed more than once in succession because to do so would only add redundant information which does not increase program complexity, the paths are

a, d, e, f, g, i, k
a, d, e, f, h, j, k
a, d, l
a, b, c, d, e, f, g, i, k
a, b, c, d, e, f, h, j, k
a, b, c, d, l

Reachability and reachability index, as defined in Table III, are also shown in Fig. 1. When reachability is high, testing is complicated because it must be demonstrated that each node can be reached from many sources. In addition, debugging is difficult because an error identified with a particular node could have many possible causes.

Experiment Results

As mentioned in the Introduction, the purpose of this research was to test the hypothesis that program structure has a significant effect on error making, detection, and correction. A related objective was the identification of complexity measures which could be used as a guide for designing programs and for allocating resources to debugging and testing. Two approaches were utilized. One approach involved determining whether a functional relationship exists between complexity measures and error properties. Sample correlation coefficients were used for this purpose. The second approach compared mean values of complexity for zero error structures with corresponding values for structures with errors.

TABLE IV
ERROR DISCOVERY WITH RESPECT TO ERROR TYPES

ERROR TYPE	NUMBER OF ERRORS Project #1	Project #2	Project #3	Project #4	TOTAL	
D3	3	3		2	8	
D7				1	1	
D9	1	10			11	Extreme Conditions
D10		1			1	
D11	1	3			4	
D12		2		5	7	
D13				1	1	
D15		1		1	2	
C1	1	1			2	
C4	1				1	
C5	1				1	
C6	1		1	2	4	
C7				1	1	
C8	2	1			3	
C9		2		1	3	
C10	1	2			3	
C11	3	2			5	
C12	4	2		1	7	
C15	1				1	
C16		1			1	
C17	1	5		3	9	
C19	1		1		2	
C20	2	1			3	
C21		2	1	2	5	
C23	6	5		1	12	Syntax
C24	2	1	1		4	
C26		1			1	
C27	2	3		4	9	
C28	5	6		6	17	Representation
C29		2			2	
C30		1			1	
A1	2	14		16	32	Manual
A2	2	1		1	4	
A3				1	1	
B3	1				1	
B4		2		1	3	

D( ): Design; C( ): Coding; A( ): Clerical; B( ): Debugging.

TABLE VI
SOFTWARE ERROR EXPERIMENT. COMPLEXITY MEASURE COMPARISON (PROCEDURES WITH NO ERRORS VERSUS PROCEDURES WITH ERRORS)

Number of Errors Found vs.		Number of Procedures
Cyclomatic Number	.78	31
Number of Source Statements	.59	31
Number of Paths	.76	20
Reachability	.77	20
Average Reachability	.78	20
Labor Time (Man-Mins) To Find Error vs.		
Cyclomatic Number	.67	31
Number of Source Statements	.59	31
Number of Paths	.90	20
Reachability	.90	20
Average Reachability	.87	20
Labor Time (Man-Mins) To Correct Error vs.		
Cyclomatic Number	.72	31
Number of Source Statements	.51	31
Number of Paths	.65	20
Reachability	.66	20
Average Reachability	.71	20

TABLE VI
SOFTWARE ERROR EXPERIMENT. COMPLEXITY MEASURE COMPARISON (PROCEDURES WITH NO ERRORS VERSUS PROCEDURES WITH ERRORS)

	No Errors		Errors	
	Mean Value	Number of Procedures	Mean Value	Number of Procedures
Cyclomatic Number	1.699	83	4.74	31
Number of Source Statements	9.361	83	27.23	31
Number of Paths	2.671	82	27.1	20
Reachability	10.1	82	120.3	20

The distribution of 173 errors by error type and project is shown in Table IV. Error types were identified in Table II. Identifications are repeated in Table IV for the more frequent error types. Manual errors (type A1) were the most frequent. These errors could result from lack of motor skill or temporary manual dysfunction involving errors of commission, omission, or transposition. The next most frequent error (type C28) is a representation error which is defined as a failure to make the correct physical representation of thoughts, such as writing a statement different from what one intended. The third most frequent error (type C23) is a syntax error in coding, involving the violation of programming language syntactical rules. The fourth most frequent error (type D9) is a program design error, extreme conditions neglected. This error occurs when extreme numeric values, which cause underflow or overflow, exceed array limits or exceed bounds, etc., are neglected.

The errors which were of interest with regard to complexity analysis were those which occurred after the first error-free compile, when the debugging phase began. Errors which were not related to structure—clerical and syntactical errors—were usually found during desk checking or compilation. Of the 173 errors found on the four projects, 64 were found during debugging and testing. These were the errors which could have been related to complexity of structure. Only four of these were clerical errors. Errors were located on the directed graphs as shown in Fig. 1.

Correlation coefficients without transformation (shown in Table V) and with log-log transformation were calculated for error properties versus complexity measures. The former were consistently higher than the latter. This was a surprising result because the relationship would appear to be nonlinear. The explanation may lie in the fact that the range of variables, 1 to 16 for cyclomatic number and 1 to 8 for number of errors, is small; the relationship may be roughly linear within a limited range. The results in Table V suggest some degree of linear association between error properties and complexity. The relationship is not strong; the existence of a continuum of paired values is not indicated. Since a functional relationship was not indicated, the next analysis involved partitioning the structures into a set with no errors and a set with errors. Mean values of complexity measures were calculated for the two sets. The results are shown in Table VI. These results are significant; all complexity measures are much higher for structures which had errors. The data which were used in the calculations of Tables V and VI are shown in Appendix D. A further substantiation of the results in Table VI is the observation that in only one case in Appendix D was a large number of paths associated with a zero error structure; in all other cases a large number of paths was associated with structures with errors.

If error properties are partitioned by high and low values of cyclomatic number the results in Table VII are obtained. This table shows significantly higher error finding and correction times for large cyclomatic numbers. Also, the bottom

TABLE VII
SOFTWARE ERROR EXPERIMENT. COMPARISON OF ERROR PROPERTIES FOR HIGH AND LOW CYCLOMATIC NUMBERS

Cyclomatic Number	Number of Errors	Total Error Finding Time (Man-Mins.)	Mean Error Finding Time (Man-Mins.)	Total Error Correction Time (Man-Mins.)	Mean Error Correction Time (Man-Mins.)
6≤V≤16 (8 structures)	31	3595	116.0	708	28.8
1≤V≤5 (23 structures)	33	1977	59.9	516	15.6

Cyclomatic Number	Number of Errors Made in		
	Design	Coding	Debugging
8, 13, 16 (4 structures)	14	7	0
1, 2, 3 (10 structures)	3	10	1

TABLE VIII
SOFTWARE ERROR EXPERIMENT. COMPARISON OF ERROR PROPERTIES BY PHASE IN WHICH ERROR MADE

	Design	Coding	Debugging
Number of Errors Made	36	131	6
Total Error Finding Time (Man-Mins.)	2937	7108	350
Mean Error Finding Time[1] (Man-Mins.)	83.9	55.5	
Total Error Correction Time (Man-Mins.)	891	1089	58
Mean Error Correction Time (Man-Mins.)	24.8	8.3	

[1] The first error in each project (one design and three coding) is not used in the calculation (35 errors used for design and 128 errors used for coding).

half of this table shows that more design errors and more coding errors are associated with high and low cyclomatic numbers, respectively. However, sample size is small in this instance.

A final pair of relationships between error properties and complexity measures is that $V \geq 5$ accounted for 40 of the 64 errors associated with structure, or 62.5 percent, and $N_p \geq 5$ accounted for 53 of the 64 errors, or 82.8 percent.

It was also of interest to analyze error relationships which do not involve program structure. One relationship is error finding and correction times as a function of phase in which an error is made. As shown in Table VIII, significantly higher error finding and correction times occurred when the errors were made in the design phase. This result seems to be related to the higher occurrence of design errors in complex structures, requiring higher error finding and correction times.

COMPARISON OF RESULTS WITH PREVIOUS WORK

This experiment was basically different from the projects described earlier because we emphasized programming and the relationship between program structure and error properties rather than analysis and design. Although some analysis and specification work was required, the major technical effort and intellectual challenge involved program design and debugging. Also there was no need for integration testing or extensive functional testing because it was not necessary to integrate and system test a large number of modules. In contrast, the results of Endres, Rubey, and Wolverton involved operating, tactical, and command and control systems where lack of understanding of machine architecture or lack of specification validity and clarity—analysis and design issues—were significant contributors to software errors. The programming orientation of the experiment is borne out by Table IX which shows that the most frequent error type is coding. This contrasts with Rubey's results of 50 percent of the errors being attributed to specification problems. Similarly, Table X shows

TABLE IX
DISTRIBUTION OF ERRORS BY MAJOR CATEGORY

Error Type	Number of Errors	Percentage
Design	35	20.2
Coding	97	56.1
Clerical	37	21.4
Debugging	4	2.3
	173	

TABLE X
DISTRIBUTION OF PROJECT EFFORT

	Man-Hours	Percentage
Design	67.0	23.9
Coding	61.5	21.9
Debugging	103.5	36.9
Testing	48.8	17.4
	280.8	

that the combined labor expenditure for coding and debugging was 59 percent, again indicating a significant involvement with programming. Wolverton's data indicates a 40-20-40 distribution among design and analysis, code and debug, and checkout and test, respectively. Since our experiment involved relatively little analysis and test (integration and system), it is natural that the labor distribution in our experiment is the inverse of Wolverton's.

Some of the differences in results are attributable to differences in error classifications and definitions of analysis, design, coding, testing, etc. We defined many more types of errors than has been the case with most projects in the past. Our error categories included not only the type of computer operation involved, e.g., indexing, but in some cases the physical or mental process (representation error) involved in error making.

The primary corroboration of our results comes from the work of McCabe [10] wherein his organization used a cyclomatic number to control program complexity, with the objective of avoiding structures which would be error prone and difficult to debug. Quantitative results were not presented in his paper, but the implication is that the use of a threshold value of complexity, which should not be exceeded, provided a valuable control for improving software reliability.

Conclusions

Based on this experiment we conclude that, for similar programming environments and assuming a stable programming personnel situation, structure would have a significant effect on the number of errors made and labor time required to find and correct the errors. This relationship is not expressible as a mathematical function. Rather, complexity measures serve to partition structures into high or low error occurrence according to whether the complexity measure values are high or low, respectively. Also complexity measures partition structures into high or low error finding and correction times according to whether the complexity measure values are high or low, respectively. We cannot say that these results would hold for all projects. However, our quantitative results correspond to qualitative results obtained by McCabe [10] in another programming environment. It would be worthwhile to use complexity measures as a program design control to discourage complex programs and as a guide for allocating testing resources. The use of complexity measures in this fashion should be tested on large production programming projects so that the hypothesis can be tested under more typical conditions.

With respect to the identification of the best complexity measure, where "best" would be the most accurate predictor or indicator of error properties, we found no best measure. Several measures–number of paths, cyclomatic number, reachability and program size–proved useful. However, the cyclomatic number has a very practical advantage–it is easy to compute. A computer program is not needed to compute the cyclomatic number; it is required for paths and reachability. Secondly, the cyclomatic number is finite, while the number of paths and reachability may be infinite, even for simple programs, if loops are present. Thirdly, the cyclomatic number has the valuable property of identifying the number of independent substructures or constructs of a structured program. Finally, for testing purposes, all circuits corresponding to substructures which should be tested can be generated once the fundamental circuits have been identified.

Appendix A

TABLE XI
WORKSHEET FOR DESIGN PHASE AND DESIGN REVIEW PHASE OF PROJECT #2

STEP #	PROBLEM AND PLANNED SOLUTION	ALTERNATE SOLUTIONS	DAY TIME	MAN HOURS /STEP	ERROR #	COMMENTS
6	Define Algorithms for Utilities to support structures using M1: - Name, Setname - Brother, Add brother - Son(i) (get name of ith son)		3/06 1800 3/06 1830	.5		
7	Define Algorithms for Utilities to support structures using M2: a) Support of linked list of path headers: - Set Path ID, -Path, ID - Linkpath (forward/backward) - Initialize Pathlist - Next path, -Previous path b) Support of path structures: - Name of (retrieve node name) - Predecessor - Successor - Number of Successors - Linking of nodes (forward and backward) - Duplicate (set duplicate node) - Add node - Duplicate a path - Implementation of alternative path		3/07 1000 3/07 1200	2.0		
8	Define Procedural Algorithms: - Remove a Path - List Path - List all Paths - Find End of a Path - Find Original		3/08 1500 1505 3/08 1530	.5	17	D9 former analysis did not include trivial case

TABLE XII
WORKSHEET FOR CODING PHASE OF PROJECT #2

Beginning of Coding (day/time): 3/06/1500

End of Coding (day/time): 3/10/1730

Man hours: 26 (including punching of cards)

CODING BEGIN DAY/TIME	CODING END DAY/TIME	PROGRAM PART	ERROR #	DAY TIME	COMMENTS
					1) Record when error is detected.
03/06/1500		-Data Definition		3/06	
		-Primitives	1	1530	C28
		-Error Handling	2	1540	C28
		-Initialization	3	1620	C23
	03/06/1630				
03/06/1630		-Punching cards			
			4	1740	C28
	03/06/1800				
03/06/1830		-Utilities			
	03/06/2030				
03/07/1200		-Utilities		3/07	
			5	1305	D9
	03/07/1400				
03/08/1000		-Punching cards (Utilities)		3/08	
			10	1000	C12 (missing BEGIN)
			11	1005	C28 (mispelling of procedure name)
			12	1005	C23
			13	1010	C16
			14	1025	C1
			15	1030	C17
			16	1030	C17
	03/08/1200				
03/08/1530		-Coding of procedural subroutines	18	1630	D9 (design did not consider removal of first and last path)
	03/08/1800				
03/08/1900		-Coding of procedural subroutines			
	03/08/2000				
03/09/2000		-Punching cards	19	2130	D11 (faultv design of index calculation)
			20	2150	C10 (faulty condition "Not Equal" instead of "=")
	03/08/2200				

TABLE XIII
WORKSHEET FOR DEBUGGING PHASE

PROJECT #: 2 DEBUG Run #: 1

Begin of Debug Run (day/time): 03/06/1800

End of Debug Run (day/time): 03/07/1400

# of Debug Steps included in Debug Run: 4 CPU time for Debug run (sec): 1.82

CPU time for necessary compiles (sec): 4.33

a) 0.89 b) 1.76 c) 1.68 d) e) f) g)

Man hours for this Debug Run: 4.0 (including preparation of debug run)

STEP #	PROGRAM PART	OBJECTIVE AND EXPECTED RESULT	ACTUAL RESULT	DAY TIME	MAN HOURS /STEP	ERROR #	COMMENTS AND CODED ERROR TYPES
							1) Record when error occurs
1	Primi-tives and part of utili-ties	Get error free compile	3 compile errors	3/06 1811 3/06 1930	1.3	5 6 7	A1 C23 A1
2		Repeat step 1	O.K.	3/07 1110	1.2		
3	Primi-tives	Check initialization, allocation and freeing of items in both freelists. Check parameter implemen-tation. Check writing of blank lines.			0.5		
		pointers of freelists must be set appropriately, allocated elements must be filled with zeroes, allocated items must be unlinked, after freeing an element becomes part of the free-list again, parameters should be set as designed)	upper half of allocated element of M2 not filled with zeroes	1250		8	A1 ("+1" left out while pun-ching cards
4		repeat steps 1 and 3	O.K.	3/07 1400	1.0		

APPENDIX B

TABLE XV
ERROR LISTING

PROJECT #: 2

Begin of Project (day/time): 03/01/1900

End of Project (day/time): 03/17/2200

Man hours for total project: 125.0

ERROR #	PHASE in which ERROR was discovered	PHASE in which ERROR was made	ERROR TYPE (see Table 2)	TIME spent to solve the ERROR (Man min.)	# of OTHER STATEMENTS OR PARTS OF THE PROGRAM AFFECTED
1	Coding	Coding	C28	5	
2	Coding	Coding	C28	5	
3	Coding	Coding	C23	2	
4	Coding	Coding	C28	1	
5	Debugging	Coding	A1	5	
6	Debugging	Coding	C23	5	
7	Debugging	Coding	A1	5	
8	Debugging	Coding	A1	10	
9	Coding	Design	D9	15	Whole algorithm affected.
10	Coding	Coding	C12	1	
11	Coding	Coding	C28	1	
12	Coding	Coding	C23	1	
13	Coding	Coding	C16	1	
14	Coding	Coding	C1	5	
15	Coding	Coding	C17	2	
16	Coding	Coding	C17	1	
17	Design	Design	D9	1	
18	Coding	Design	D9	30	Whole algorithm affected.
19	Coding	Design	D11	5	
20	Coding	Coding	C10	5	
21	Debugging	Coding	C23	5	
22	Debugging	Coding	C17	15	
23	Debugging	Coding	C27	5	
24	Debugging	Coding	A1	2	
25	Debugging	Coding	C17	5	
26	Debugging	Coding	A2	10	
27	Debugging	Coding	C28	10	
28	Debugging	Coding	C17	5	
29	Debugging	Coding	C11	5	
30	Deugging	Coding	A1	15	

TABLE XIV
TEST PHASE DESCRIPTION

Project #: 2

Test run #: 1 Including 1 Test Step

Begin of Test (day/time): 3/17/1500 End of Test (day/time): 03/17/1900

CPU time for necessary compiles (in sec.): 7.53

a) 7.53 b) c) d) e) f) g)

CPU time for TEST run (sec): 25.29

Man Hours for this Test run: 3.0 (including preparation of tests)

TEST STEP	OBJECTIVE	EXPECTED RESULT (TOLERANCE)	ACTUAL RESULT	ERROR #	DAY TIME	COMMENTS AND CODED ERROR TYPES
1	Test performance of program for various directed graphs:				3/17 1500	1) Record when error occurs.
	a) directed graph 9 nodes, 10 arcs	3 valid paths	O.K.			
	b) directed graph 6 nodes, 7 arcs	3 valid paths	O.K.			
	c) directed graph 6 nodes, 7 arcs	4 valid paths	O.K.			
	d) directed graph 22 nodes, 25 arcs	7 valid paths	O.K.			
	e) directed graph 22 nodes, 27 arcs	11 valid paths	O.K.			
	f) directed graph 1 node, 1 arc (self loop at node 1)	no valid path warning should be printed	O.K.			
	g) directed graph 2 nodes, 2 arcs	2 valid paths	O.K.			
	h) directed graph 2 nodes, 1 arc	1 valid path	O.K.		3/17 1900	

Remarks: Directed graphs submitted for testing include a variety of boundary conditions and special cases which are considered to be difficult to examine.

APPENDIX C

TABLE XVI
ERROR LISTING (COMMENTS)

ERROR #	DAY TIME	COMMENTS (EVIDENCE, THOUGHTS, WHY WAS THE ERROR MADE? WHY AND HOW WAS THE ERROR DISCOVERED? ERROR BLOCKING, etc.)
	03/06	
1	1530	Errors 1,2, 3 were discovered while reading previously written sections of code.
2	1540	
3	1620	
4	1740	Lack of concentration while punching cards. (Main disadvantage while punching cards is that punched data is not immediately seen after each key stroke.)
5	1815	same as 4
6	1815	Programmer did not check programming manual. (error could have been avoided)
7	1815	same as 4
	03/07	
8	1305	
9	1305	
	03/08	
10	1000	Errors 10-17 were detected because programmer punched cards himself. It is quite natural that he uses this time to review his code.
11	1005	
12	1005	
13	1010	
14	1025	
15	1030	
16	1030	
17	1505	
18	1630	Error found during desk test.
19	2130	Error found while punching cards.
20	2150	same as 19
21	2200	
22	2200	Function name used as local variable.
23	2200	
24	2200	
25	2245	Incomplete correction of error #22.
26	2310	Mandatory declaration omitted while punching cards.
27	2323	Right parenthesis omitted.
	03/09	
28	1030	Lack of concentration while punching cards. (Programmer was tired.)
29	1030	same as 28
30	1030	same as 28
31	1030	same as 28
32	1030	same as 28

APPENDIX D

TABLE XVII
SOFTWARE ERROR EXPERIMENT. COMPLEXITY MEASURES VERSUS ERROR PROPERTIES FOR PROCEDURES WITH ONE OR MORE ERRORS

	Complexity Measures					Error Properties		
Project/ Procedure	Np	V	R	r	S	e	$\sum T_f$ (Man-mins)	$\sum T_c$ (Man-mins)
1/1	2	2	7	1.4	14	1	35	10
1/5	*	6	*	*	26	5	95	53
1/6	*	5	*	*	7	2	37	35
1/8	*	5	*	*	21	1	35	15
1/9	2	2	8	1.333	6	1	115	20
2/1.19	1	1	2	1.0	3	1	10	10
2/1.23	1	1	2	1.0	11	1	110	10
2/2.2	2	1	4	1.0	8	1	15	10
2/7	3	2	7	1.4	15	3	230	45
2/9	72	8	370	19.47	45	3	140	185
2/10	9	4	25	2.778	18	1	10	5
2/11	*	6	*	*	54	3	950	65
2/12	5	2	13	1.444	34	2	300	30
2/15	*	4	*	*	19	1	5	1
2/16	*	5	*	*	30	2	150	20
2/18	12	4	50	2.941	26	1	5	15
2/21	*	16	*	*	94	8	750	145
3/3	2	2	6	1.5	13	1	60	5
4/7	40	6	151	3.438	83	1	120	5
4/13	16	5	64	4.267	28	1	20	15
4/14	*	8	*	*	37	5	255	65
4/15	4	3	12	2.0	13	2	40	35
4/21.3	7	3	34	4.857	16	1	0	5
4/22	*	7	*	*	34	1	160	30
4/23	18	5	60	4.615	24	1	125	10
4/27	5	4	23	2.091	18	3	360	120
4/28	*	5	*	*	35	2	90	50
4/29	321	13	1468	54.37	49	5	1125	160
4/30	6	4	24	2.4	19	1	60	30
4/31	*	4	*	*	27	1	30	10
4/33	14	4	76	7.6	17	2	135	10

*Very large value.

TABLE XVIII
SOFTWARE ERROR EXPERIMENT. COMPLEXITY MEASURES FOR PROCEDURES WITH ZERO ERRORS

Project/ Procedure	Np	V	R	r	S
1/2	3	2	8	1.333	6
1/3.1	1	1	2	1.0	8
1/3.2	1	1	2	1.0	11
1/3.3	1	1	2	1.0	4
1/4	6	3	26	4.333	18
1/7	7	3	40	5.0	15
2/1.1	1	1	2	1.0	3
2/1.2	1	1	2	1.0	3
2/1.3	1	1	2	1.0	3
2/1.4	1	1	2	1.0	3
2/1.5	1	1	2	1.0	3
2/1.6	1	1	2	1.0	3
2/1.7	1	1	2	1.0	3
2/1.8	1	1	2	1.0	3
2/1.9	1	1	2	1.0	5
2/1.10	1	1	2	1.0	5
2/1.11	1	1	2	1.0	5
2/1.12	1	1	2	1.0	13
2/1.13	1	1	2	1.0	3
2/1.14	1	1	2	1.0	3
2/1.15	1	1	2	1.0	3
2/1.16	1	1	2	1.0	3
2/1.17	1	1	2	1.0	3
2/1.18	1	1	2	1.0	3
2/1.20	1	1	2	1.0	3
4/1.1	1	1	2	1.0	2
4/1.2	1	1	2	1.0	2
4/1.3	1	1	2	1.0	7
4/1.4	1	1	2	1.0	5
4/1.5	1	1	2	1.0	7
4/1.6	1	1	2	1.0	5
4/1.7	1	1	2	1.0	5
4/1.8	1	1	2	1.0	5
4/1.9	1	1	2	1.0	5
4/1.10	1	1	2	1.0	4
4/1.11	1	1	2	1.0	3
4/1.12	1	1	2	1.0	3
4/1.13	1	1	2	1.0	3
4/1.14	1	1	2	1.0	3
4/1.15	1	1	2	1.0	3
4/1.16	1	1	2	1.0	3
4/1.17	1	1	2	1.0	3
4/1.18	1	1	2	1.0	3
4/1.19	1	1	2	1.0	5
4/1.20	1	1	2	1.0	5
4/1.21	1	1	2	1.0	6
4/1.22	1	1	2	1.0	9
4/1.23	1	1	2	1.0	6
4/2.1	2	1	4	1.0	8
4/2.2	2	1	4	1.0	9
4/2.3	2	1	4	1.0	9
4/3	2	2	6	1.5	4
4/4.1	2	2	8	1.6	7
4/4.2	2	2	8	1.6	9
4/5	8	4	38	4.222	56
4/6	4	1	6	1.0	24
4/8.1	2	2	8	1.6	13
4/8.2	2	2	8	1.6	13
4/8.3	2	2	8	1.6	10
4/8.4	2	2	8	1.6	9
4/8.5	2	2	8	1.6	12
4/9	16	5	95	7.917	21
4/10	12	5	65	4.643	49
4/11	7	3	38	5.429	19
4/12	*	4	*	*	20
4/16.1	2	2	6	1.5	6
4/16.2	2	2	6	1.5	12
4/16.3	2	2	6	1.5	9
4/16.4	2	2	6	1.5	10
4/17	16	1	18	1.0	21
4/18	8	4	42	3.818	21
4/19	4	3	16	2.667	11
4/20	3	2	9	1.125	13
4/21.1	7	3	34	4.857	14
4/24	16	7	83	4.368	19
4/25.1	2	2	8	1.333	15
4/25.2	2	2	8	1.333	10
4/25.3	2	2	8	1.333	17
4/26	3	3	15	1.5	19
4/32	7	3	36	5.143	15
4/34	2	2	5	1.25	15

*Very large value.

REFERENCES

[1] J. Goldberg, Ed., in *Proc. Symp. on the High Cost of Software*, 1973.

[2] E. W. Dijkstra, "Notes on structured programming," in *Structured Programming*. New York, NY: Academic, 1972, ch. I, pp. 1–82.

[3] G. H. Bradley, T. F. Green, G. T. Howard, and N. F. Schneidewind, "Structure and error detection in computer software," in *Proc. AIEE Conf.*, 1975, pp. 54–59.

[4] T. F. Green, N. F. Schneidewind, G. T. Howard, and R. Pariseau, "Program structures, complexity and error characteristics," in *Proc. Symp. on Comput. Software Eng.*, 1976, pp. 139–154.

[5] N. F. Schneidewind, "The use of simulation in the evaluation of software," *Computer*, pp. 47–53, Apr. 1977.

[6] H.-M. Hoffmann, "An experiment in software error occurrence and detection," Masters thesis, Naval Postgraduate School, Monterey, CA, June 1977.

[7] A. B. Endres, "An analysis of errors and their causes in systems programs," *IEEE Trans. Software Eng.*, vol. SE-1, no. 2, pp. 140–149, June 1975.

[8] R. J. Rubey, J. A. Dana, and P. W. Biche, "Quantitative aspects of software validation," *IEEE Trans. Software Eng.*, vol. SE-1, no. 2, pp. 150–155, June 1975.

[9] R. W. Wolverton, "The cost of developing large scale software," *IEEE Trans. Comput.*, vol. C-23, no. 6, pp. 615–636, June 1974.

[10] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[11] S.-P. Chan, *Introductory Topological Analysis of Electrical Networks*. New York, NY: Holt, Rinehart, and Winston, 1969, chs. 1–2, pp. 1–83.

N. F. Schneidewind (A'54–M'59–M'72–SM'77) received degrees in electrical engineering, operations research, and management from the University of California and the University of Southern California.

He held various technical management positions with System Development Corporation, Planning Research Corporation, Computer Usage Company, and UNIVAC. He is presently a Professor of Computer Science and Administrative Sciences at the Naval Postgraduate School, Monterey, CA, where he teaches courses in real-time systems, systems analysis and design, and operating systems. His current research involves software engineering and computer networks.

Dr. Schneidewind is a member of Sigma Xi, Eta Kappa Nu, and Alpha Pi Mu, and is an Officer of the IEEE Technical Committee on Simulation and the ACM Special Interest Group on Software Engineering.

Heinz-Michael Hoffmann received the M.S. degree in computer science (with distinction) from the United States Naval Postgraduate School. He also received a degree from the Federal German Naval Academy.

For many years he has been involved in the development of computer systems for tactical data systems for the Naval Command and Control Systems Command and has been an Instructor for cadets aboard a training ship. He is presently a Lieutenant Commander in the Federal German Navy, Flensburg, Germany.